The Implementation of Lua 5.0

Roberto Ierusalimschy[1], Luiz Henrique de Figueiredo[2], Waldemar Celes[1]

[1]Department of Computer Science, PUC-Rio, Rio de Janeiro, Brazil.

[2]IMPA–Instituto de Matemática Pura e Aplicada, Rio de Janeiro, Brazil.

摘要 我们探讨 Lua 5.0 的主要特点：基于寄存器的虚拟机、散列表用做数组时的新优化算法，闭包的实现以及协程的加入。

1. 介绍

Lua 作为一个内部的软件开发工具，诞生于学院实验室，然而，不知何故，很快被全世界的几个工业项目所采用，而且目前正被广泛运用于游戏行业。[1]

如何看待 Lua 被广泛使用的这种现象？我们认为，答案在于对 Lua 的设计和实现目标：提供一种嵌入式的脚本编程语言，它简洁、高效、可移植并且是轻量级的。自 1993 年 Lua 诞生以来，这些就一直是我们主要的目标，而且随着 Lua 的演变，这个目标一直被我们所追求。[2]

Lua 广泛的用途催生了对语言特性的需求。Lua 的多种特性已经被工业需求和用户反馈所提出。在 Lua 5.0 中引入协程以及在即将发布的 Lua 5.1 版中实现增量式垃圾回收就是重要的例证。

本文中，我们探讨与 Lua 4.0 相比，Lua 5.0 的主要的特点。

基于寄存器的虚拟机：传统上，大部分虚拟机都是基于堆栈的。这种形势自 Pascal 的 P-虚拟机开始一直持续到今天的 Java 虚拟机（JVM ）以及 Microsoft .Net 环境。 然而近来，基于寄存器的虚拟机正日益激起人们的兴趣（例如，规划中的 Perl6（Parrot）虚拟机将会是基于寄存器的）。据我们所知，Lua 5.0 的虚拟机是第一个广泛使用的基于寄存器的虚拟机。该虚拟机将在第 7 节中介绍。

散列表用做数组时的新优化算法：不同于其他脚本编程语言，Lua 不提供数组类型。取而代之的是，Lua 程序员用表和整数索引来实现数组。Lua5.0 用一种

---

[1] 2003 年一个由重量级的游戏开发网站 gamedev.net 发起的一项非正式调查显示，Lua 是游戏开发行业最受欢迎的脚本语言。更详细的信息请浏览网页 http://www.gamedev.net/gdpolls/viewpoll.asp?ID=163。

[2] 采用了类 MIT 版权许可证也有助于 Lua 的推广。

新的算法来检测散列表是否被用作数组并且自动地将与数值索引相关的值存储在一个实际的数组中，而不是将其加入散列表。这个算法在第四节中讨论。

闭包的实现：Lua5.0 支持带作用域的嵌套函数。该机制给那些用一个堆栈来存储活动记录的语言出了个大难题。Lua 用一种新颖的方法来实现函数闭包。函数闭包将局部变量保存在（基于数组的）栈中，只有当某局部变量在被嵌套函数引用期间超出作用域时，它们才被移入堆中。

协程的加入：Lua5.0 在语言中引入协程的概念。虽然协程的实现多少有些老套，不过出于完整性的考虑，我们还是会在第六节对它做个简介。

其他各小节会详细论述以上各项内容，第二节我们对 Lua 的设计目标以及此目标如何影响语言的实现做一个总览。第三节介绍在 Lua 内部，值是如何表示的。虽然值的表示方法本身没什么新颖之处，但在其他小节我们需要这一内容。最后，第八节将给出一个小的对比测试结果并得出某些结论。

2. Lua 的设计和实现总览

正如介绍中提到的，我们实现 Lua 的目标是：

简洁：我们寻求最简化的语言和最小化的源码（以 C 语言实现）。这意味着 Lua 只有一些类似传统编程语言的简单的语法和少量的语言结构。

高效：我们希望快速编译并执行 Lua 程序。这意味着需要一个快速、智能、一遍编译的编译器和一个快速执行指令的虚拟机。

可移植：我们希望 Lua 能够在尽可能多的平台上运行。希望 Lua 内核能够在不做任何修改的情况下，在任何平台上都能顺利编译通过。并且希望 Lua 程序在任何平台上都不需要修改就能顺利执行，只要该平台上有一个 Lua 解释器。这意味着需要用纯 ANSI C 实现 Lua 并注意移植问题，避开 C 语言及其库的阴暗面，并保证在 C++编译器上也能顺利通过编译，而不希望看到警告信息。

可嵌入：Lua 是一种可扩展的语言，它的设计目标是在大型程序中提供脚本支持。这些目标都需要一个形式简单、功能强大，但依赖于 C 语言内建数据类型的编程接口（C API）。

容易嵌入：我们希望能够很容易地将 Lua 嵌入到应用程序中，而不会显得臃肿。这意味着需要紧凑的 C 代码和小巧的 Lua 内核，同时可通过用户库扩展 Lua 的功能。

这些目标有些矛盾。例如，Lua 常常被用作数据描述语言，用于存取配置文件，有时甚至是大型数据库（数 MB 的 Lua 程序并非罕见）。这就需要一个快速的 Lua 编译器。另一方面，我们也希望 Lua 程序能快速执行。这需要编译器够聪明，能够为虚拟机生成好的指令代码。因此，Lua 编译器的实现必须权衡这两种需求。然而，编译器不可能太庞大；否则将导致整个 Lua 内核源码的膨胀。目前编译器的代码量占 Lua 内核的 30%左右。对于内存受限的应用场合，如嵌入式系统，有可能只嵌入 Lua 虚拟机而不需要编译器。Lua 程序在离线状态被预编译完成而在运行期被微系统载入执行（这时由于载入的是字节码，因此执行速度同样很快）。

Lua 早期版本都是使用手工编写的词法分析器，和手工编写的递归下降语法分析器。直到 Lua3.0 版开始使用由 YACC 自动生成的语法分析器。当语言的语法结构经常变化（不再稳定）时，使用 YACC 能很好地适应这种语法的变化。但手写的分析器更小巧、更高效、更可移植，并完全地可重入，也能提供更好的出错信息。

Lua 编译器不产生语法树的中间表示（IR）。虽然它在读取并分析 Lua 源程序的同时就生成了虚拟机指令。然而，仍然会执行一些优化。例如，对于像变量和常数这样的基础表达式的代码生成会延迟。当分析这类表达式时，暂时不产生指令码，而是用一种简单的结构表示它们。这样，就能轻易地检测出某个指令的操作数是否是常数或局部变量，从而决定是否要在指令中直接使用这些值，进而避免产生不必要的、费时的数据拷贝 MOVE 指令(见第 3 节）。

为了保证可以在各种不同的 C 编译器和平台上顺利移植，Lua 不能使用类似 direct threaded code[8]一类常用于解释器中的各种小技巧。Lua 使用标准的 while-switch 分派循环。偶尔有些地方的 C 代码看起来过于复杂，但这是为了保证可移植性。伴随着 Lua 在不同平台（包括一些 64 位平台和一些 16 位平台）的各种 C 编译器下顺利通过编译，其可移植性多年来持续稳定。

我们认为已经达到了这些既定的目标。Lua 是一种高度可移植的语言：Lua 运行在各类有 ANSI C 编译器的平台上，从嵌入式系统到大型机。Lua 真的是轻量级的：在 Linux 上 Lua 解释器和完整的标准库总体小于 150KB；Lua 内核小于 100KB。Lua 是高效的：对比测试表明，Lua 是脚本语言（如解释型语言和动态类型语言）家族里执行速度最快的。也可以认为 Lua 是一门简洁的语言，语法类似 Pascal，语义有点像 Scheme，当然，这只是我们自己的观点。

3. 值的内部表示

Lua 是动态类型的语言：类型是与值而不是与变量相关。Lua 有 8 种基本的值类型：nil，boolean，number，string，table，function，userdata 和 thread。nil 类型是个标记类型，只有一种值，就是 nil。Boolean 类型的值有 true 和 false 两种值。number 类型是双精度的浮点数，对应于 C 语言中的 double，但这不是绝对的，可以通过重新编译 Lua 来将其重设为 float 或 long 型。（一些游戏终端和小型机缺乏支持 double 数据类型的硬件。）string 是字节数组，有一个显式的长度，因此可以容纳任何二进制数，包括 0。table 是关联数组，可以通过任何（除了 nil）值来索引，也能容纳任意值。function 可以是 Lua 函数或根据 Lua 虚拟机接口函数的原型编写的 C 函数。Userdata 实际上是一个指向用户内存块的指针，分两种情况：heavy，内存由 Lua 分派，并由垃圾回收机制负责处理，light，内存由用户分配并释放。最后是 thread 类型，它代表协程。任何类型的值都是 first-class 的:可以将其存入全局变量、局部变量或 table 域中，或作为实际参数传递给函数，或从函数中返回值，等等。

Lua 将值表示成带标志的联合结构，即，（t，v）对，其中 t 是个整数，代表值 v 的类型，v 是一个 C 语言 union 类型数据结构，它存储有实际的值。 nil 型只有单个值。boolean 和 number 实现为未包装的值：v 直接对应于 union 中由 t 指示的域。这意味着 union 必须有足够空间容纳一个 double 型。string，table，function，thread 和 userdata 型数据通过引用来实现：v 中含有一个指向结构的指针，该结构实现由 t 指定的类型。这些结构共用一个头结构，头结构中含有垃圾回收所需的信息。结构的剩余部分包含的信息对应于指定的数据类型。

图一显示了实现 Lua 值类型的代码片段，TObject 是实现 Lua 值类型的主要

结构：它代上述带标志的联合(t,v)对。Value 是实现 Lua 值的联合类型。nil 类型不需要在 union 中明确表示出来，因为 t 标志已经足够说明 nil 值了。Value 的 n 域用于表示 number（默认情况下，lua_Number 被定义成 double 类型）。b 域用于 boolean。p 域用于 light userdata。gc 域用于需要垃圾回收机制处理的其他值（如 string，table，function，heavy userdata，threads 等）。

```
typedef struct {
  int t;
  Value v;
} TObject;
```

```
typedef union {
  GCObject *gc;
  void *p;
  lua_Number n;
  int b;
} Value;
```

Figure 1: Lua values are represented as tagged unions.

使用带标志的联合结构来表示Lua值导致的一个直接结果就是拷贝Lua值比较费时：在 double 值为 64 位的 32 位机器上，TObject 是 12 字节（或 16 字节，如果机器以 8 字节边界对齐 double 的话），因此拷贝一个值需要拷贝 3 或 4 个机器字，然而，在 ANSI C 范围内，很难再更好地表示一个 Lua 值了。一些动态类型语言（如 Smalltalk80 的早期实现）使用每个指针多余的位来存储值的类型标志。这种技巧在常规的机器上可以工作，因为出于边界对齐的原因，指针的最后 2-3 位总是 0，因此可以利用。然而这种技术既不可移植又不能由 ANSI C 实现。C 语言标准甚至不保证指针是一个整体，因此没有对指针进行位操作的标准方法。

另一种减少值所占空间的方法是保留值的类型标志，但不将 double 放入 union 中。例如，所有 number 都可以表示成在堆上分配的对象，就像 string 一样。（Phyon 用这种技术，除了它预先分配一些小的整数值。）然而，这种表示方法导致语言十分低效。还有一种办法是将整数值保留，而将浮点数放入堆中，但这种方法将大大增加实现数学运算的复杂性。

就像早期的解释型语言，如 Snobol [11]和 Icon [10]一样，Lua 用一个散列表将 string 内部化：Lua 为每个字符串只保留一份拷贝，而且字符串是不变的：一旦内部化，字符串将不可更改。字符串的散列值由一个结合了位运算和数学运算的简单表达式计算出来，计算过程中会对所有数据位进行随机洗牌。当字符串内部化时，散列值被保存起来，以便后面的字符串比较和表索引操作能快速进行。

如果字符串太长，那么散列函数就不再逐个考察每个字节，因而能快速计算长字符串的散列值。避免长字符串处理时的性能损失是非常重要的，因为在 Lua 中，计算长字符串的散列值是很普遍的。例如，在 Lua 中经常将整个文件读入一个长字符串中进行处理。

4. 表

表是 Lua 中主要---实际上是唯一---的表示数据结构的工具。表不仅在 Lua 语言中，而且在 Lua 的实现中都扮演关键角色。表被用于完成许多内部任务，为了无需担忧性能问题，我们努力以一种好的方法实现表结构，这花费了我们很多时间。同时我们也得到了回报。漂亮的实现方法使得整个语言规模保持足够精简。反过来说，缺乏其他数据构造机制增加了高效实现表结构的难度。

Lua 中的表是关联数组，即可以通过任何值（除了 nil）来索引表项，表项可以存储任何类型的值。此外，表是动态的，当有数据加入其中（对不存在的表项赋值），或从中移除数据（将 nil 赋给表项）时，它们可以自动伸缩。

不同于其他脚本语言，Lua 中没有内置对数组类型的支持。数组是用表和整数索引来模拟的。用表来模拟数组有助于语言的实现。这样做主要是出于简单性：Lua 不需要两套截然不同的指令来处理表和数组。此外，程序员也不需要在这两种表示之间进行选择。在 Lua 中实现稀疏数组价值不大，例如，在 Perl 中，如果你试图执行程序：$a[1000000000]=1;可能导致内存不足，因为这会导致 Perl 创建一个拥有 10 亿个元素的数组，而等价的 Lua 程序，a={[1000000000]=1}，创建的表只有一个表项。

截至 Lua4.0 版，表都是严格地以散列表（哈希表）实现的：所有的键、值都明确地存在于表中。在 Lua5.0 中，对表被用作数组的情形使用了一种新的算法来进行优化：对于键是整数的表项，将不保存键，只将值存入一个真正的数组中。更准确地说，在 Lua5.0 中，表以一种混合型数据结构来实现，它包含一个散列表部分和一个数组部分。对于键、值对 "x"→9.3, 1→100, 2→200, 3→ 300，图 2 展示了一种可能的形式。注意右边的数组部分：它不保存整数键。只有在底层实现时才需要注意这点区别，其他情况下，即使是对虚拟机来说，访问表项也

是由底层自动统一操作的，因而用户不必考虑这种区别。表会根据其自身的内容自动动态地使用这两个部分：数组部分试图保存所有那些键介于 1 和某个上限 n 之间的值。非整数键和超过数组范围 n 的整数键对应的值将被存入散列表部分。

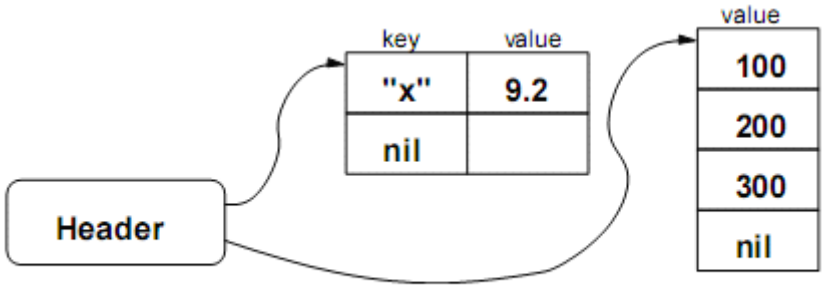


Figure 2: A Lua table.

当表需要增长时，Lua 重新计算散列表部分和数组部分的大小。最初表的两个部分有可能都是空的。新的数组部分的大小是满足以下条件的最大的 n 值：1 到 n 之间至少一半的空间会被利用（避免像稀疏数组一样浪费空间）；并且 n/2+1 到 n 之间的空间至少有一个空间被利用（避免 n/2 个空间就能容纳所有数据时申请 n 个空间而造成浪费）。当新的大小计算出来后，Lua 为数组部分重新申请空间，并将原来的数据存入新的空间。举例来说，假设 a 是一个空表，散列表部分和数组部分都是 0 大小。如果执行 a[1]=v,那么表就需要增长以容纳新键。Lua 会选择 n=1 作为新数组的大小（存储一个数据 1→v）。散列表部分仍保持为空。

这种混合型结构有两个优点。第一，存取整数键的值很快，因为无需计算散列值。第二，也是更重要的，相比于将其数据存入散列表部分，数组部分大概只占用一半的空间，因为在数组部分，键是隐含的，而在散列表部分则不是。结论就是，如果表被当作数组用，只要其整数键是紧凑的（非稀疏的），那么它就具有数组的性能，而且无需承担散列表部分的时间和空间开销，因为这种情况下散列表部分根本就不存在。相反，如果表被当作关联数组用，而不是当数组用，那么数组部分就可能不存在。这种内存空间的节省很重要，因为在 Lua 程序中，常常创建许多小的表，例如，当用表来实现对象时。

散列表部分是用链散表结合文献[3]中 Brent 介绍的变异方法实现的。这种表的主要确定性是：如果一个元素不在其主位置上（例如，由散列值给出的原始位置），那么冲突元素就会在这个主位置上。换句话说，只有在两个元素拥有同样

的主位置时才会出现冲突。（例如，俩元素对同一个表大小，有同样的散列值。）由于不存在次级冲突，这种表的负载因子可以达到 100%而没有任何性能损失。

5. 函数和闭包

当 Lua 编译一个函数时，会生成一个原型。该原型包含有函数的虚拟机指令、常数值（数值、字符串等），以及一些调试信息。在运行期，任何时候只要 Lua 执行一个 function...end 表达式，它就会创建一个新的闭包。每个闭包都有一个对函数原型的引用、一个对环境的引用（环境其实是一个表，函数可在该表中索引全局变量，后面细述），和一个数组，数组中每个元素都是一个对 upvalue 的引用，可通过该数组来存取外层的局部变量。

作用域（生存期）规则下的嵌套函数给如何实现内层函数存取外层函数的局部变量出了个众所周知的难题。考虑图 3 的例子。当 add2 被调用时，其函数体存取外层局部变量 x（Lua 中，函数参数也是局部变量）。然而，此时创建 add2 的函数 add 已经返回了。如果是 x 在栈中分配的，此刻 x 已经不存在了（x 的存储空间已经被回收了）。

```
function add (x)                    add2 = add(2)
  return function (y)               print(add2(5))
           return x+y
         end
end
```

Figure 3: Access to outer local variables.

许多过程式语言，通过限制作用域（如 Python），或限制函数嵌套（如 Pascal），或限制两者（如 C 语言）来避开这个难题，函数式语言没有这些限制。对于像 Scheme 和 ML 这样的非纯函数式语言的研究已经积累了大量关于闭包的编译技术（见文献[18,1,20]）[3]。然而这些研究都不考虑编译器的复杂程度。例如，仅仅为了完成 Bigloo 的数据流分析一项，Scheme 的优化编译器就比整个 Lua 实现复杂 10 倍：Bigloo2.6f 的 Cfa（数据流分析）模块的源码有 10635 行，而 Lua 核心代码才 10155 行。正如第二节解释的，Lua 需要一种更简单的办法。

[3] 像 Haskell 语言这样的纯函数式语言所用的技术通常不适用于过程式语言。

Lua 用一种称为 upvalue 的结构来实现闭包。对任何外层局部变量的存取间接地通过 upvalue 来进行。upvalue 最初指向栈中变量活跃的地方（图 4 左边）。当离开变量作用域时（超过变量生存期时），变量被复制到 upvalue 中（图 4 右边）。由于对变量的存取是通过 upvalue 里的指针间接进行的，因此复制动作对任何存取此变量的代码来说都是没有影响的。与内层函数不同的是，声明该局部变量的函数直接在堆栈中存取它的局部变量。

通过为每个变量至少创建一个 upvalue 并按所需情况进行重复利用，保证了未决状态（是否超过生存期）的局部变量（pending vars）能够在闭包间正确地共享。为了保证这种唯一性，Lua 为整个运行栈保存了一个链接着所有正打开着的 upvalue（那些当前正指向栈内局部变量的 upvalue）的链表（图 4 中未决状态的局部变量的链表）。当 Lua 创建一个新的闭包时，它开始遍历所有的外层局部变量，对于其中的每一个，若在上述 upvalue 链表中找到它，就重用此 upvalue，否则，Lua 将创建一个新的 upvalue 并加入链表中。注意，一般情况下这种遍历过程在探查了少数几个节点后就结束了，因为对于每个被内层函数用到的外层局部变量来说，该链表至少包含一个与其对应的入口（upvalue）。一旦某个关闭的 upvalue 不再被任何闭包所引用，那么它的存储空间就立刻被回收。

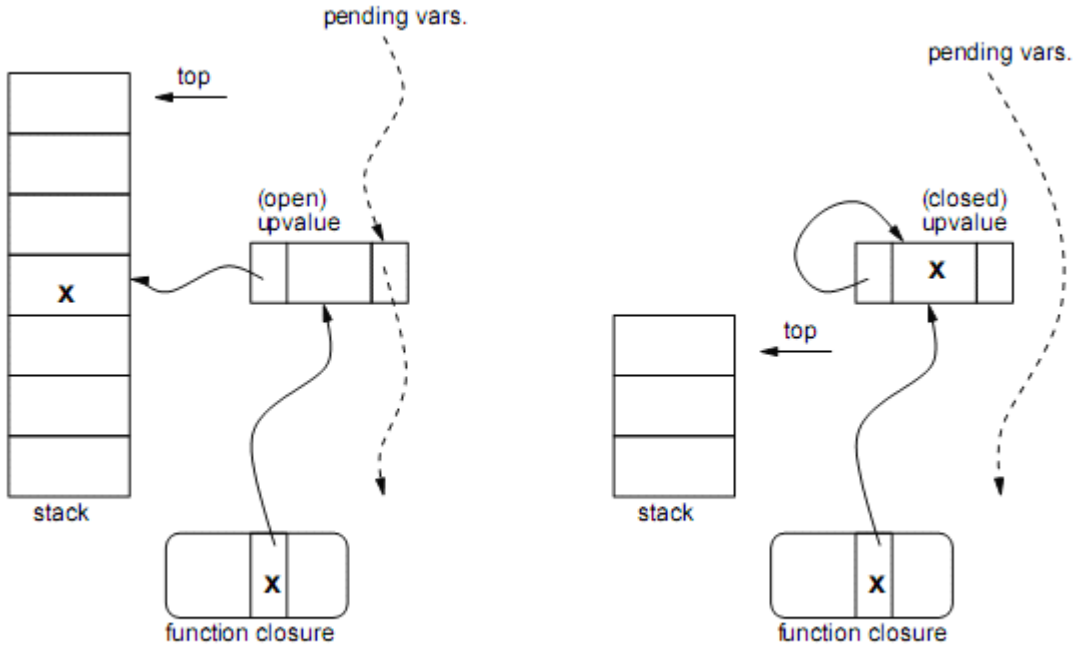


Figure 4: An upvalue before and after being "closed".

一个函数有可能存取其更外层函数而非直接外层函数的局部变量。这种情况

下，有可能当闭包创建时，此局部变量尚不存在。Lua 使用 flat 闭包来处理这种情况。有了 flat 闭包，无论何时只要函数存取更外层的局部变量，该变量也会进入其直接外层函数的闭包中。这样，当一个函数被实例化时，所有进入其闭包的变量就在直接外层函数的栈或闭包中了。

6. 线程和协同程序（协程）

从 5.0 版开始，Lua 实现不对称协程（也称为半不对称协程或不完全协程）。三个 Lua 标准库函数提供了对协程的支持：create，resume 和 yield。（这些函数位于 coroutine 名字空间中。）create 函数接收“主”函数并用此函数创建一个新的协程。它返回一个代表协程的值，该值具有 thread 类型。（与 Lua 中任何其他值一样，协程也是 first-class 的）。resume 函数让指定协程（重新）开始执行，进而调用其主函数。yield 函数暂停执行中的协程 B，并将其控制权返还给之前调用 resume 使 B 得以执行的协程 A。

概念上讲，每个协程都有各自的栈。（具体说，每个协程有两个栈，正如我们第七节会讲到的，但我们可以把它们看成是一个抽象栈）。Lua 中协程是有栈的，这样我们就可以在多级函数嵌套调用内挂起（暂停执行）一个协程。解释器只是简单地将整个栈放在一边而在另一个栈上继续执行。一个程序可以任意重启任何挂起的协程。当与栈相关的协程不可用时，垃圾回收器就回收栈空间。

协程的有栈性和 first-class 特征，正如中 Lua 实现的，使得它们具有独特的可扩展性。这样就允许程序员用协程来实现多种高级的控制机制，如协作式多线程，生成器，对称协程，回溯等[7]。

在 Lua 中，实现协程的时候，一个关键点是解释器不能用其自身的 C 工作栈来实现要被解释执行的函数调用指令。（Python 社团将满足这种限制条件的解释器称为无栈需求的解释器[22]。）当解释器程序的主循环解释执行一个调用指令时，会在（协程的运行）栈中开辟一个新的区块并调整几个指针，然后继续执行主循环去执行被调用的函数。类似地，返回指令移除栈顶的区块，调整指针，然后继续执行主循环去执行调用者的后续指令。这并非巧合，在真实的 CPU 中正是这样执行函数调用的。

当执行 resume 时，解释器会递归地调用自身的主循环函数去执行恢复运行的协程，并利用此协程的栈实现调用和返回指令。当协程执行 yield 调用时，将返回到前一个对解释器主循环函数的调用点，从而不再执行当前协程的后续指令。换句话说，任意时刻，Lua 都用 C 堆栈跟踪活动协程的栈。每次对 yield 的调用解释器都返回到上次执行 resume 的地方。

在某些程序语言中，实现协程的困难之处源于如何处理对外层局部变量的引用。因为一个协程中正在执行的函数可能是由另一个协程创建的，而该函数有可能引用另一个栈中的变量。这引出了一种被称为“仙人掌”的结构。flat 闭包的使用，正如我们在第 5 节讨论过的，完全避免了这个难题。

7. 虚拟机

Lua 解释器在执行 Lua 程序时，首先将源码编译成虚拟机指令（opcode，操作码），然后执行这些指令。对每一个被编译的函数，Lua 为其创建一个原型，原型中含有一个由该函数的虚拟机指令组成的数组、一个所有被该函数用到的常数值（TObjects，字符串或实数）的数组（译者注：这很重要，因为这避免了在指令码中直接包含常数值进而导致指令长度的膨胀。事实上，可以把这些常数看成具有只读属性的全局变量，对它们的处理和全局变量的处理是一致的）。

在十年的时间里（从 1993 年 Lua 发布开始），各种版本的 Lua 都使用基于堆栈的虚拟机。从 2003 年开始，随着 Lua5.0 的发布，Lua 改用个基于寄存器的虚拟机。新的虚拟机也用堆栈分配活动记录，寄存器就在该活动记录中。当进入 Lua 程序的函数体时，函数从栈中预分配一个足以容纳该函数所有寄存器的活动记录。函数的所有局部变量都各占据一个寄存器。因此，存取局部变量是相当高效的。

使用寄存器式虚拟机消除了用堆栈式虚拟机时为了在栈中拷贝数据而必需要的大量出入栈（push/pop）指令。在 Lua 中，这些出入栈指令相当费时，因为它们需要拷贝带标志的值（tagged value, TObject）正如第三节讨论过的。因此寄存器结构既消除了昂贵的值拷贝操作，又减少了为每个函数生成的指令码数量。Davis al.[6]反对基于寄存器的虚拟机，并以了 Java 虚拟机的字节码作为反证。某

些编译器作者也因为可以很容易在编译期间为堆栈式虚拟机生成代码而反对寄存器式虚拟机。

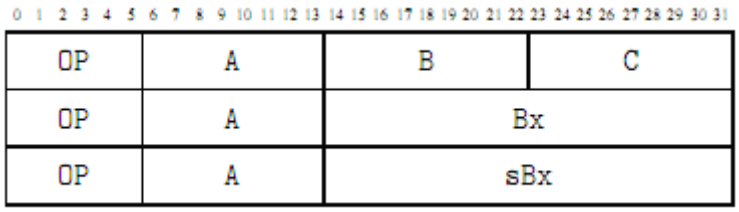
与寄存器式虚拟机相关的两个难题是：指令大小和译码速度。寄存器式虚拟机的指令需要指明操作数位置，因此通常要比堆栈式虚拟机的同类指令长。（例如，当前 Lua 虚拟机的指令长度是 4 字节，而其他许多典型的堆栈式虚拟机的指令长度只有 1-2 字节，包括前一版本的 Lua 也是。）另一方面，为基于寄存器的虚拟机生成的操作码要比堆栈式虚拟机少，因此指令总长度大不了多少。

堆栈式虚拟机的许多指令都有隐含的操作数。而寄存器式虚拟机中对应的指令需要从其中解码出操作数。解码过程增加了解释器的负担。有几个因素会淡化这种负面影响，第一，堆栈式虚拟机也花费一些时间处理隐含的操作数（例如，增减栈指针）。第二，由于寄存器式虚拟机中所有操作数都在指令中，而指令是一个机器字，对操作数的解码过程只包含一些很廉价的操作，如逻辑运算。另外，堆栈式虚拟机的指令常常需要多字节操作数。如 Java 虚拟机 JVM 的跳转和分支指令用了两字节的偏移量。出于对齐的关系，解释器无法一次获取这样的操作数（至少对于可移植的代码来说是如此，因为它必须总是假定最坏对齐的限制条件）。在寄存器式虚拟机上，由于操作数在指令里面，解释器无需单独获取它们。

Lua 虚拟机有 35 条指令。大部分的指令直接对应着语言结构：数学运算、表的创建和索引、函数和方法调用、存取变量值等。也有一套用于实现控制结构的常规跳转指令。图 5 列出了全部指令及其简短用法。一些符号的意义如下：R（X）是第 X 个寄存器；K（X）是第 X 个常数；RK（X）是 R（X）或 K（X-k），取决于 X 的值---当 X<k（一个内置参数，一般是 250）时，取 R（X）。G[X]是全局变量表的 X 域。U[X]是第 X 个 upvalue。若要看关于 Lua 的虚拟机指令的完整讨论请参考[14,21]。

寄存器保存在运行栈中，它们实质是一个数组。因此存取寄存器是很快的。常数和全局变量也存于不同数组中，因此存取它们也很快。全局变量表是一个普通的 Lua 表，虽然要通过散列法来存取其域，但其实是很高效的，因为它是由（对应于全局变量名的）字符串来索引的，而字符串散列值是已被预先计算出来了的，这一点在第二节已提到过。

Lua 的虚拟机指令将 32 位分成 3 或 4 个区域，如下。OP 域是指令，占 6 位。其他域是操作数。A 域总是存在的，占 8 位。B 域和 C 域各占 9 位。它们可以组合成一个 18 位的 Bx 域（无符号）或 sBx（有符号）。




```
MOVE        A B      R(A) := R(B)
LOADK       A Bx     R(A) := K(Bx)
LOADBOOL    A B C    R(A) := (Bool)B; if (C) PC++
LOADNIL     A B      R(A) := ... := R(B) := nil
GETUPVAL    A B      R(A) := U[B]
GETGLOBAL   A Bx     R(A) := G[K(Bx)]
GETTABLE    A B C    R(A) := R(B)[RK(C)]
SETGLOBAL   A Bx     G[K(Bx)] := R(A)
SETUPVAL    A B      U[B] := R(A)
SETTABLE    A B C    R(A)[RK(B)] := RK(C)
NEWTABLE    A B C    R(A) := {} (size = B,C)
SELF        A B C    R(A+1) := R(B); R(A) := R(B)[RK(C)]
ADD         A B C    R(A) := RK(B) + RK(C)
SUB         A B C    R(A) := RK(B) - RK(C)
MUL         A B C    R(A) := RK(B) * RK(C)
DIV         A B C    R(A) := RK(B) / RK(C)
POW         A B C    R(A) := RK(B) ^ RK(C)
UNM         A B      R(A) := -R(B)
NOT         A B      R(A) := not R(B)
CONCAT      A B C    R(A) := R(B) .. ... .. R(C)
JMP         sBx      PC += sBx
EQ          A B C    if ((RK(B) == RK(C)) ~= A) then PC++
LT          A B C    if ((RK(B) <  RK(C)) ~= A) then PC++
LE          A B C    if ((RK(B) <= RK(C)) ~= A) then PC++
TEST        A B C    if (R(B) <=> C) then R(A) := R(B) else PC++
CALL        A B C    R(A), ... ,R(A+C-2) := R(A)(R(A+1), ... ,R(A+B-1))
TAILCALL    A B C    return R(A)(R(A+1), ... ,R(A+B-1))
RETURN      A B      return R(A), ... ,R(A+B-2)      (see note)
TFORLOOP    A C      R(A+2), ... ,R(A+2+C) := R(A)(R(A+1), R(A+2));
TFORPREP    A sBx    if type(R(A)) == table then R(A+1):=R(A), R(A):=next;
SETLIST     A Bx     R(A)[Bx-Bx%FPF+i] := R(A+i), 1 <= i <= Bx%FPF+1
SETLISTO    A Bx
CLOSE       A        close stack variables up to R(A)
CLOSURE     A Bx     R(A) := closure(KPROTO[Bx], R(A), ... ,R(A+n))
```

Figure 5: The instructions in Lua's virtual machine.

大部分指令使用三地址格式，其中 A 代表目的寄存器；B，C 分别代表源操作数，可能是寄存器或常数（使用前面提到的 RK（X）表示法）。按照这种格式，

Lua 中许多典型的操作都可以编码到一条指令中。例如，对一个局部变量进行增量运算，如 a=a+1 被编码成 ADD x x y，其中 x 代表局部变量 a 所占的寄存器，y 代表常数 1。当 a，b 都是局部变量时，像 a=b.f 这样的赋值也可以编码成一条指令 GETTABLE x y z，其中 x 是 a 所占的寄存器，y 是 b 所占的寄存器，z 是字符串常量"f"的索引（记住常量都在常量数组中，而不是在指令中）。

分支指令的实现有点困难，因为每个分支指令需要给出两个用来做比较的操作数，还需要一个跳转偏移量。将这些都编码到一条指令中将限制跳转距离在 256 之内（假设用 9 位的有符号整数域 B 或 C 做偏移量）。Lua 采用的解决办法是这样的：概念上讲，当分支指令的测试条件失败时，只跳过后续一个指令；被跳过的指令是一个无条件跳转指令，它用 18 位的偏移量。实际上，由于一个分支指令后总是有一个跳转指令，解释器会将这两条指令一起执行。也就是说，当一个分支指令的测试条件为真时，解释器立即取回出下一条指令并执行跳转，而不会等到下一个分派周期（while-switch 循环）。图 6 显示了一个 Lua 源码和字节码的例子。注意观察上述无条件跳转指令紧随分支跳转指令的情况。

```
function max (a,b)
  local m = a           1 MOVE     2 0 0   ; R(2) = R(0)
  if b > a then         2 LT       0 0 1   ; R(0) < R(1) ?
    m = b               3 JMP      1       ; to 5 (4+1)
  end                   4 MOVE     2 1 0   ; R(2) = R(1)
  return m              5 RETURN   2 2 0   ; return R(2)
end                     6 RETURN   0 1 0   ; return
```

Figure 6: Bytecode for a Lua function.

```
local a,t,i             1: LOADNIL  0 2 0
a=a+i                   2: ADD      0 0 2
a=a+1                   3: ADD      0 0 250 ; 1
a=t[i]                  4: GETTABLE 0 1 2
```

Figure 7: Register-based opcode (Lua 5.0).

图 7 展示了一个 Lua 编译器执行优化的小例子。图 8 展示了同样的源码由 Lua4.0 编译输出的指令，Lua4.0 的虚拟机是基于堆栈的，有 49 条指令。注意到，寄存器式虚拟机允许编译器生成更短的指令序列。源码中的每条可执行语句在 Lua5.0 编译器下都被编译成一条指令，但在 Lua4.0 中却需要 3 或 4 条指令。

```
local a,t,i             1: PUSHNIL    3
a=a+i                   2: GETLOCAL   0     ; a
                        3: GETLOCAL   2     ; i
                        4: ADD
                        5: SETLOCAL   0     ; a
a=a+1                   6: GETLOCAL   0     ; a
                        7: ADDI       1
                        8: SETLOCAL   0     ; a
a=t[i]                  9: GETLOCAL   1     ; t
                       10: GETINDEXED 2     ; i
                       11: SETLOCAL   0     ; a
```

Figure 8: Stack-based opcode (Lua 4.0).

Lua 采用寄存器窗来应对函数调用。Lua 从寄存器窗中第一个未使用的寄存器开始，连续存放实际参数的值。当执行调用时，这些存放有实际参数的寄存器成为被调用函数的活动记录的一部分，因此被调用函数就能像对待常规局部变量一样存取实参。当函数返回时，实参被放回调用者的活动记录中。

Lua 用两个并列的栈实现函数调用。（实际上，每个协程各自都有两个这样的栈，正如我们在第六节讨论的。）其中一个栈为每个活动着的函数保存一个入口，该入口存有被调用函数(的原型指针)、返回地址、以及一个基址（指针），该索引指向被调用函数的活动记录。另一个栈只是一个大的值数组，用来保存那些活动记录。每个活动记录都存有函数的临时变量（参数，局部变量等）。事实上我们可以看到，对于第一个栈中的每个入口，在第二个栈中都有一个与之对应的、大小不定的入口。

8. 结论

我们介绍了 Lua5.0 最具创新性的方面：基于寄存器的虚拟机，表当做数组用时新的优化算法以及闭包的实现。

据我们所知，Lua 是第一个被广泛使用的、采用寄存器式虚拟机的脚本语言。对表的优化措施允许表在被用作数组时被部分实现为真实的数组（即当数组对于键 1..n 有足够多元素空间）。闭包的实现方法也是独一无二的，结合使用了基于数组的栈和支持作用域规则的嵌套函数，而没有复杂性的控制流分析。

图 9 的表展示了 Lua 新老版本性能对比得出的一些简单结果。对比测试在一台 512MB RAM、Linux2.6 操作系统的奔腾 4 机器上进行。Lua 由 gcc3.3 编译。

Lua4.0 既不用基于寄存器的虚拟机（用的是堆栈式虚拟机），也不进行表的数组式优化。Lua5.0’版类似 Lua5.0 版，只是没有进行表的数组式优化，和尾递归调用、（与协程相关的）动态栈；Lua5.0'版实际上是一个拥有寄存器式虚拟机的 Lua4.0 版。

我们根据 “计算机语言大比拼”[2]上的测试项进行了全面的测试，除了第一个（sum）测试项。sum 测试项只是一个简单的从整数 1 加到 n 的循环。sum 测试项花去了虚拟机的大部分时间，结果表明新的虚拟机是老版的基于堆栈的虚拟机的速度的 2 倍以上。其他任务（函数调用、表/数组索引）上的测试没有这么大的性能提升。因此平均来说，虚拟机自身的性能提升对全部测试项总体的运行时间影响较小。不过在使用数组（筛选、堆排序、矩阵运算）的测试项中，使用新的虚拟机和对数组操作进行优化两项可以在总体上节约 40%的执行时间。

可以在 http://www.lua.org/source/5.0/下载到 Lua5.0 的完整源码。

program	Lua 4.0	Lua 5’	Lua 5.0
sum (2e7)	1.23	0.54 (44%)	0.54 (44%)
fibo (30)	0.95	0.68 (72%)	0.69 (73%)
ack (8)	1.00	0.86 (86%)	0.88 (88%)
random (1e6)	1.04	0.96 (92%)	0.96 (92%)
sieve (100)	0.93	0.82 (88%)	0.57 (61%)
heapsort (5e4)	1.08	1.05 (97%)	0.70 (65%)
matrix (50)	0.84	0.82 (98%)	0.59 (70%)

Figure 9: Benchmarks (times in seconds; percentages are relative to Lua 4.0).

感谢：

Edgar Toernig 为闭包的实现提供了极其重要的建议. Thatcher Ulrich 实现了 Lua4.0 的协程机制，这直接影响了我们对 Lua5.0 的协程机制的实现。本文的作者由 CNPq（302608/2002-8, 300392/2003-6 和 401109/2003-8）、FINEP (CT-INFO 01/2003)和 Microsoft Research (2nd Rotor RFP)提供部分支持。

参考文献：

1. A. W. Appel. Empirical and analytic study of stack versus heap cost for languages with closures. Journal of Functional Programming, 6(1):47C74 1996.

2. D. Bagley. The great computer language shootout. http://www.bagley.org/~doug/shootout/.

3. R. P. Brent. Reducing the retrieval time of scatter storage techniques. Communications of the ACM, 16(2):105C109 1973.
4. A. Calpini. The great Win32 computer language shootout. http://dada.perl.it/shootout/.
5. L. Cardelli. Compiling a functional language. In LISP and Functional Programming, pages 208C217 1984.
6. B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron. The case for virtual register machines. In Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators, pages 41C49 ACM Press, 2003.
7. A. de Moura, N. Rodriguez, and R. Ierusalimschy. Coroutines in Lua. Journal of Universal Computer Science, 10(7):910C925 July 2004.
8. M. A. Ertl and D. Gregg. The structure and performance of ecient interpreters. Journal of Instruction-Level Parallelism, 5:1C25 Nov. 2003.
9. A. Goldberg and D. Robson. Smalltalk-80: the language and its implementation. Addison-Wesley, 1983.
10. R. E. Griswold and M. T. Griswold. The Implementation of the Icon Programming Language. Princeton University Press, 1986.
11. R. E. Griswold, J. F. Poage, and I. P. Polonsky. The SNOBOL 4 Programming Language. Prentice-Hall, 1971.
12. R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The evolution of an extension language: A history of Lua. In Proceedings of V Brazilian Symposium on Programming Languages, pages BC14CBC 2001.
13. S. C. Johnson. YACC: Yet another compiler compiler. CS TR 32, Bell Labs, July 1975.
14. K.-H. Man. A no-frills introduction to Lua 5 VM instructions. http://luaforge.net/docman/?group_id=83.
15. S. Pemberton and M. Daniels. Pascal Implementation: The P4 Compiler and Interpreter. Ellis Horwood, 1982.
16. I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 291C300 Montreal, Canada, June 1998.
17. A. Randal, D. Sugalski, and L. Toetsch. Perl 6 and Parrot Essentials. OReill,

second edition, 2004.
18. M. L. Scott. Programming Language Pragmatics. Morgan Kaufmann, 2000.
19. M. Serrano. Control ow analysis: a functional language compilation paradigm. In 10th ACM Symposium on Applied Computing, pages 118C122 Nashville, TN, Feb. 1995.
20. M. Serrano and P. Weis. Bigloo: A portable and optimizing compiler for strict functional languages. In 2nd Static Analysis Symposium, pages 366C381 Glasgow, Scotland, Sept. 1995. LNCS 983.
21. Z. Shao and A. W. Appel. Space-ecient closure representations. In ACM Conference on Lisp and Functional Programming, pages 150C161 June 1994.
22. Z. A. Shaw. The Lua virtual machine.
http://www.zedshaw.com/writings/luas-lvm-instructions/luas_lvm_
instructions/luas_lvm_instructions.html.
23. C. Tismer. Continuations and stackless Python. In Proceedings of the 8th International Python Conference, Arlington, VA, 2000.
24. P. Winterbottom and R. Pike. The design of the Inferno virtual machine.
http://www.herpolhode.com/rob/hotchips.html.

译者声明：

感谢巴西人为世界奉献如此美妙的程序语言！感谢云风为中国的程序员们大力推荐了这门语言！感谢每位阅读了这篇译文的朋友们！

翻译本文的目的是为了学习 Lua 并与他人分享和交流学习心得，因此不追求格式上和原文统一，而且标题和原文作者没有翻译，省略了参考文献。文中图片没有专门绘制，而是直接采自原文，我相信读者不会感到陌生。

如果文中有任何您认为翻译不准确或误导的地方，请直接参考原文，也欢迎您批评指正。

我的邮箱是：xianghaifei@sohu.com